A Short Introduction to the *OmicsMarkeR* Package

Charles Determan Jr.*

August 20, 2015

1 Introduction

The *OmicsMarkeR* package contains functions to streamline the analysis of 'omics' level datasets with the objective to classify groups and determine the most important features. *OmicsMarkeR* loads packages as needed and assumes that they are installed. I will provide a short tutorial using the both synthetic datasets created by internal functions as well as the 'Sonar' dataset.

Install *OmicsMarkeR* using

```
source("http://bioconductor.org/biocLite.R")
biocLite("OmicsMarkeR")
```

to ensure that all the needed packages are installed.

*cdetermanjr@gmail.com

2 Basic Classification Example

OmicsMarkeR has a few simplified functions that attempt to streamline the classification and feature selection process including the addition of stability metrics. We will first generate a synthetic dataset. This includes three functions that can be used to create multivariate datasets that can mimic specific omics examples. This can include a null dataset via the `create.rand.matrix` to create a random multivarate dataset with `nvar = 50` and `nsamp = 100`. The `create.corr.matrix` function induces correlations to the datasets. The `create.discr.matrix` function induces variables to be discriminate between groups. The number of groups can be specified with `num.groups`.

```
library("OmicsMarkeR")
set.seed(123)
dat.discr <- create.discr.matrix(
    create.corr.matrix(
        create.random.matrix(nvar = 50,
                             nsamp = 100,
                             st.dev = 1,
                             perturb = 0.2)),
    D = 10
)
```

To avoid confusion in the coding, one can isolate the variables and classes from the newly created synthetic dataset. These two objects are then used in the `fs.stability` function. I can then choose which algorithm(s) to apply e.g. `method = c("plsda", "rf")`, the number of top important features `f = 20`, the number of bootstrap reptititions for stability analysis `k = 3`, the number of k-fold cross-validations `k.folds = 10` and if I would like to see the progress output `verbose = TRUE`.

warning: You will receive warnings if you run code exactly as shown in this vignette. This is intentional as the PLSDA runs with this simple dataset often only need a single component but you need to indicate 2 to fit the model. This is provided for the users information.

```
vars <- dat.discr$discr.mat
groups <- dat.discr$classes
fits <- fs.stability(vars,
                     groups,
                     method = c("plsda", "rf"),
                     f = 10,
                     k = 3,
                     k.folds = 10,
                     verbose = 'none')
```

```
## foreach:  simple, scalable parallel programming from Revolution Analytics
## Use Revolution R for scalability, fault tolerance and more.
## http://www.revolutionanalytics.com
```

```
## Warning:  package 'plyr' was built under R version 3.2.1
```

```
## randomForest 4.6-10
## Type rfNews() to see new features/changes/bug fixes.

## Warning: package 'e1071' was built under R version 3.2.1

## Loading required package: survival

## Warning: package 'survival' was built under R version 3.2.1

## Loading required package: lattice

## Warning: package 'lattice' was built under R version 3.2.1

## Loading required package: splines
## Loading required package: parallel
## Loaded gbm 2.1.1
## Loading required package: cluster

## Warning: package 'cluster' was built under R version 3.2.1

## Loading required package: Matrix

## Warning: package 'Matrix' was built under R version 3.2.1

## Loaded glmnet 2.0-2

## Warning in training(data = trainData, method = method[i], tuneValue = as.data.frame(best
: PLSDA model contained only 1 component.
##                PLSDA requires at least 2 components.
##
## Model fit with 2 components

## Warning in training(data = trainData, method = method[i], tuneValue = as.data.frame(best
: PLSDA model contained only 1 component.
##                PLSDA requires at least 2 components.
##
## Model fit with 2 components

## Warning in training(data = trainData, method = method[d], tuneValue = tuned.methods$best
: PLSDA model contained only 1 component.
##                PLSDA requires at least 2 components.
##
## Model fit with 2 components

## Warning in training(data = trainData.new[[d]], method = method[d], tuneValue = as.data.f
: PLSDA model contained only 1 component.
##                PLSDA requires at least 2 components.
##
## Model fit with 2 components

## Warning in training(data = trainData, method = method[d], tuneValue = tuned.methods$best
: PLSDA model contained only 1 component.
```

```
##                PLSDA requires at least 2 components.
##
## Model fit with 2 components

## Warning in training(data = trainData.new[[d]], method = method[d], tuneValue = as.data.f
:  PLSDA model contained only 1 component.
##                PLSDA requires at least 2 components.
##
## Model fit with 2 components
```

If I would like to see the performance metrics, I can simply use the `performance.metrics` function. This will provide a concise data.frame of confusion matrix and ROC statistics. Additionally, the Robustness-Performance Trade-off value (RPT) is provided with the results.

```
performance.metrics(fits)
```

```
##                         plsda         rf
## Accuracy           0.88333333 0.93333333
## Kappa              0.76666667 0.86666667
## ROC.AUC            0.92000000 0.96000000
## Sensitivity        0.86666667 0.93333333
## Specificity        0.90000000 0.93333333
## Pos Pred Value     0.90303030 0.93333333
## Neg Pred Value     0.87777778 0.93333333
## Accuracy SD        0.07637626 0.11547005
## Kappa SD           0.15275252 0.23094011
## ROC.AUC SD         0.07549834 0.06928203
## Sensitivity SD     0.11547005 0.11547005
## Specificity SD     0.10000000 0.11547005
## Pos Pred Value SD  0.10013765 0.11547005
## Neg Pred Value SD  0.10715168 0.11547005
```

```
fits$RPT
```

```
##     plsda        rf
## 0.7933472 0.3017964
```

If I would want to see the occurance of the features the model identified as the most important, this is accomplished by `feature.table`. This function returns a simple table reporting the consistency (i.e. how many times identified) and frequency (percent identified in all runs).

```
feature.table(fits, "plsda")
```

```
##    features consistency frequency
## 1        V1           3         1
## 2       V11           3         1
## 3       V14           3         1
## 4       V21           3         1
```

```
## 5       V23           3         1
## 6       V26           3         1
## 7       V28           3         1
## 8       V37           3         1
## 9       V16           2     0.667
## 10       V2           1     0.333
## 11      V47           1     0.333
## 12      V41           1     0.333
## 13      V44           1     0.333
```

If the user is interesting in applying the fitted model (determined by `fs.stability`) towards some new data this can be accomplished with `predictNewClasses`. This could either be yet another level to evaluate the tuned model's performance or if the user is applying the model in a production type setting where you are systematically using this model on new data that comes in.

```
# create some 'new' data
newdata <- create.discr.matrix(
    create.corr.matrix(
        create.random.matrix(nvar = 50,
                             nsamp = 100,
                             st.dev = 1,
                             perturb = 0.2)),
    D = 10
)$discr.mat

# original data combined to a data.frame
orig.df <- data.frame(vars, groups)

# see what the PLSDA predicts for the new data
# NOTE, newdata does not require a .classes column
predictNewClasses(fits, "plsda", orig.df, newdata)
```

Note - This function is not as efficient as I would like at present. Currently it requires the original dataset to refit the model from the parameters retained from `fs.stability`. I intend to provide the option to retain fitted models so a user can simply pull them without a need to refit. However, I am concerned about the potential size of objects (e.g. random forest, gbm, etc.). Thoughts and contributions are welcome.

3 Ensemble Methods

In machine learning ensembles of models can be used to obtain better predictive performance than any individual model. There are multiple types of ensemble types including the bayes optimal classifier, bayesian model averaging (BMA), bayesian model combination (BMC), boostrap aggregation (bagging), and boosting. Although it is the intention to include all the methods the only currently implemented method is bagging.

Bagging, in the simplest terms, is defined as giving a set of trained models equal weight when 'voting' on an optimal solution. The most familiar application of this concept is in the random forest algorithm. This technique requires a defined aggregation technique to combine the set of models. Implemented methods include Complete Linear Aggregation (`CLA`), Ensemble Mean (`EM`), Ensemble Stability (`ES`), and Ensemble Exponential (`EE`).

To conduct an ensemble analysis, the code is nearly identical to the first example in Section 1. The `fs.ensembl.stability` function contains the same arguments as `fs.stability` in addition to a few more for the ensemble components. Please see `?fs.ensemble.stability` for complete details on each. The two major additional parameters are `bags` and `aggregation.metric`. The `bags` parameter naturally defines the number of bagging iterations and the `aggregation.metric` is a string defining the aggregation method. These arguments have common defaults predefined so the call can be:

```
fits <- fs.ensembl.stability(vars,
                             groups,
                             method = c("plsda", "rf"),
                             f = 10,
                             k = 3,
                             k.folds = 10,
                             verbose = 'none')
```

As in Section 1, the `performance.metrics` function can be applied for summary statistics.

3.1 Aggregation Methods

If the user wishes to apply an aggregation method manually utilizing results from an alternative analysis, this can also be done. This package provides a the wrapper `aggregation` to apply this analysis. For these methods, the variables must have been ranked and the rownames assigned as the variable names. The function then will return that aggregated list of variables with their new respective ranks.

```
# test data
ranks <- replicate(5, sample(seq(50), 50))
row.names(ranks) <- paste0("V", seq(50))

head(aggregation(ranks, "CLA"))
```

```
##     [,1]
## V45  1.0
## V7   2.0
## V50  3.0
## V18  4.0
## V15  5.0
## V20  6.5
```

This is used internally in `fs.ensembl.stability` to optimize which variables are selected to be included in the final optimized model. The only exception to the format above is with the `EE` function where the number of variables must be defined with `f`.

4 Custom Tuning

The default implementation assumes that each model will be tuned with the same resolution of tuning parameters. For example, a default call with PLSDA and Random Forest will result in tuning 3 components and 3 mtry for each respectively. However, let's say I want to be more fine tuned with my random forest model. You can create a customized grid using `denovo.grid`.

```
# requires data.frame of variables and classes
plsda <- denovo.grid(orig.df, "plsda", 3)
rf <- denovo.grid(orig.df, "rf", 5)

# create grid list
# Make sure to assign appropriate model names
grid <- list(plsda=plsda, rf=rf)

# pass to fs.stability or fs.ensemble.stability
fits <- fs.stability(vars,
                     groups,
                     method = c("plsda", "rf"),
                     f = 10,
                     k = 3,
                     k.folds = 10,
                     verbose = 'none',
                     grid = grid)
```

The user can create their own grid completely manually but must use the appropriate names as defined by the functions. These can be check with `params`. As an example: `paramsmethod="plsda"`.

Note - the argument names must be preceded by a period. This is to prevent any unforseen conflicts in the code.

5 Stability Metrics

5.1 Stability Metric Basics

It is quite possible that a user may already have fitted a model previously or using a model that has not yet been implemented in this package. However, they may be interested in applying one or more of the stability metrics defined within this package. These functions are very simple to use. To demonstrate, let's create some sample data consisting of our **Metabolite Population**.

```
metabs <- paste("Metabolite", seq(20), sep="_")
```

Now, let's say you have run you different special model twice on different samples of a dataset. You complete your feature selection and get two lists of Metabolites. Here I am just randomly sampling.

```
set.seed(13)
run1 <- sample(metabs, 10)
run2 <- sample(metabs, 10)
```

The user can now evaluate how similar the sets of metabolites selected are via multiple possible stability metrics. These include the Jaccard Index (`jaccard`), Dice-Sorensen (`sorensen`), Ochiai's Index (`ochiai`), Percent of Overlapping Features (`pof`), Kuncheva's Index (`kuncheva`), Spearman (`spearman`), and Canberra (`canberra`). The latter two methods are not *Set Methods* and do require the same number of features in each vector. The relevant citations are provided in each function's documenation.

The general use for most of the functions is:

```
jaccard(run1, run2)
```

```
## [1] 0.25
```

The exception to this is Kuncheva's Index. This requires one additional parameter, the number of features (e.g. metabolites) in the original dataset. This metric is designed to account for smaller numbers of variables increasing the likelihood of matching sets by chance. Naturally, if you have many more variables it would be far more indicative of significance if you see the same small subset again and again as opposed to a small set seeing the same variables.

```
# In this case, 20 original variables
kuncheva(run1, run2, 20)
```

```
## [1] 0.4
```

5.2 Pairwise Stability

5.2.1 Pairwise Feature Stability

The above examples immediately lead to the question, what if I have more than two runs? What I have 3, 5, 10, or more bootstrap iterations? This is also know as a data perturbation ensemble approach. It would be very tedious to have to call the same function for every single comparison. Therefore a pairwise function exists to allow a rapid comparison between all sets. This `pairwise.stability` is very similar to the individual stability functions in practice. Let's take an example consisting of 5 runs.

```
set.seed(21)
# matrix of Metabolites identified (e.g. 5 trials)
features <- replicate(5, sample(metabs, 10))
```

Please note that currently only *matrix* objects are accepted by `pairwise.stability`. To use the function, you simply pass your matrix of variables and stability metric (e.g. sorensen). The only exception is when applying Kuncheva's Index where the `nc` parameter again must be set (which can be ignored otherwise). This will return all list containing the upper triangular matrix of stability values and an overall average.

```
pairwise.stability(features, "sorensen")
```

```
## $comparisons
##            Resample.2 Resample.3 Resample.4 Resample.5
## Resample.1        0.4        0.4        0.5        0.5
## Resample.2        0.0        0.5        0.5        0.5
## Resample.3        0.0        0.0        0.5        0.5
## Resample.4        0.0        0.0        0.0        0.7
##
## $overall
## [1] 0.5
```

5.2.2 Pairwise Model Stability

Now, in the spirit of this package, you may have the alternate approach whereby you have created several bootstrapped data sets and run a different statistical model on each data set. You could compare each one manually, but again to avoid such tedious work another function is provided for specifically this purpose. Let's take a theoretical example where I have bootstrapped 5 different data sets and applied two models to each dataset (PLSDA and Random Forest). Please note that currently only *list* objects are accepted by `pairwise.model.stability`.

Note - here I am only randomly sampling but in practice the each model would have been trained on the same dataset.

```
set.seed(999)
plsda <-
    replicate(5, paste("Metabolite", sample(metabs, 10), sep="_"))
rf <-
    replicate(5, paste("Metabolite", sample(metabs, 10), sep="_"))

features <- list(plsda=plsda, rf=rf)

# nc may be omitted unless using kuncheva
pairwise.model.stability(features, "kuncheva", nc=20)
```

```
## $comparisons
##             Resample.1 Resample.2 Resample.3 Resample.4 Resample.5
## plsda.vs.rf        0.3        0.2        0.5        0.3        0.5
##
## $overall
## [1] 0.36
```

6 Permuation Analysis

One additional level of analysis often applied to these datasets is Monte Carlo Permuations. For example, I would like to check the chance that my data can distinguish between the groups by chance. This can be done by permuting the groups in the dataset and applying the model on each permuation. This can be accomplished with `perm.class`, which also provides a plot of the classification distribution. Additionally, one may be interested in another way to evaluate the importance of variables to the distinguishing the groups. Once again, groups can be permuted, the model refit to the data and the importance of the variables evaluated. This is accomplished with `perm.features`.

```
# permuate class
perm.class(fits, vars, groups, "rf", k.folds=5,
           metric="Accuracy", nperm=10)


# permute variables/features
perm.features(fits, vars, groups, "rf",
        sig.level = .05, nperm = 10)
```

7 Parallel Analysis

Given the repetitive nature of this analysis there are ample opportunities to level the power of parallel computing. These include `fs.stability`, `fs.ensembl.stability`, `perm.class`, and `perm.features` simply by specifying the parameter `allowParallel = TRUE` in the respective function. However, the parallel backend must be registered in order to work. There are slight differences between operating systems so here are two examples.

For Unix OS, you probably will use *doMC*

```
library(doMC)

n <- detectCores()
registerDoMC(n)
```

For a Windows OS, you likely with use the *doSNOW*

```
library(parallel)
library(doSNOW)

# get number of cores
n <- detectCores()

# make clusters
cl <- makeCluster(n)

# register backend
registerDoSNOW(cl)
```

NOTE - remember to stop your clusters on Windows when you are finished with `stopCluster(cl)`.

```
sessionInfo()
```

```
## R version 3.2.0 (2015-04-16)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 7 x64 (build 7601) Service Pack 1
##
## locale:
## [1] LC_COLLATE=English_United States.1252  LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252 LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] parallel  splines   stats     graphics  grDevices utils     datasets  methods
## [9] base
##
## other attached packages:
##  [1] caTools_1.17.1      glmnet_2.0-2        Matrix_1.2-2        pamr_1.55
##  [5] cluster_2.0.3       gbm_2.1.1           lattice_0.20-33     survival_2.38-3
##  [9] e1071_1.6-6         randomForest_4.6-10 DiscriMiner_0.1-29  plyr_1.8.3
## [13] foreach_1.4.2       OmicsMarkeR_1.1.1   devtools_1.8.0
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.0          class_7.3-13         gtools_3.5.0         digest_0.6.8
##  [5] chron_2.3-47         R6_2.1.0             BradleyTerry2_1.0-6  stats4_3.2.0
##  [9] evaluate_0.7.2       httr_1.0.0           ggplot2_1.0.1        highr_0.5
## [13] BiocInstaller_1.18.1 biocViews_1.36.0     curl_0.9.1           caret_6.0-52
## [17] data.table_1.9.4     SparseM_1.6          minqa_1.2.4          car_2.0-25
## [21] nloptr_1.0.4         RUnit_0.4.28         assertive_0.3-0      proto_0.3-10
## [25] lme4_1.1-8           stringr_1.0.0        RCurl_1.95-4.7       munsell_0.4.2
## [29] compiler_3.2.0       rversions_1.0.2      BiocGenerics_0.14.0  mgcv_1.8-7
## [33] nnet_7.3-10          codetools_0.2-14     BiocCheck_1.4.0      XML_3.98-1.3
## [37] permute_0.8-4        brglm_0.5-9          MASS_7.3-43          bitops_1.0-6
## [41] grid_3.2.0           RBGL_1.44.0          nlme_3.1-121         assertive.base_0.0-1
## [45] gtable_0.1.2         git2r_0.10.1         magrittr_1.5         formatR_1.2
## [49] scales_0.2.5         graph_1.46.0         stringi_0.5-5        reshape2_1.4.1
## [53] getopt_1.20.0        optparse_1.3.0       xml2_0.1.1           BiocStyle_1.6.0
## [57] iterators_1.0.7      tools_3.2.0          Biobase_2.28.0       pbkrtest_0.4-2
## [61] colorspace_1.2-6     memoise_0.2.1        knitr_1.11           quantreg_5.11
```